Reified Type Parameters Using Java Annotations

Prodromos Gerakios Aggelos Biboudis Yannis Smaragdakis

Department of Informatics
University of Athens
{pgerakios,biboudis,smaragd}@di.uoa.gr

Abstract

Java generics are compiled by-erasure: all clients reuse the same bytecode, with uses of the unknown type erased. `C++` templates are compiled by-expansion: each type-instantiation of a template produces a different code definition. The two approaches offer tradeoffs on multiple axes. We propose an extension of Java generics that allows by-expansion translation relative to selected type parameters only. This language design allows sophisticated users to get the best of both worlds at a fine granularity. Furthermore, our proposal is based on Java 8 Type Annotations (JSR 308) and the Checker Framework as an abstraction layer for controlling compilation without changes to the internals of a Java compiler.

Categories and Subject Descriptors D.1.5 [*Programming techniques*]: Object-oriented programming; D.3.4 [*Programming languages*]: Processors—Code generation; D.3.2 [*Programming languages*]: Language Classifications—Extensible languages

Keywords mixins, reification, type annotation, pluggable types

1. Introduction

Java generics are compiled via the technique of *type erasure* or just *erasure*: Type parameters are removed by the compiler and replaced by their bound (`Object` or other, if specified). The generated bytecode contains no generic information and all instantiations share a single classfile. This approach satisfies the crucial requirement of backward compatibility and avoiding alterations of the JVM specification. The erasure technique also succeeds in lowering the code generation burden: the generic's code is not replicated on every type-instantiation. Type erasure has often been criticised in the research literature and developer communities. Firstly, the compiler inserts type casts where erasure happened to ensure compatibility with the JVM. Furthermore, data structures instantiated by primitive types like `int`, must suffer from the cost of boxing. The greatest issue, however, is the limited capabilities of reflective operations, as type parameter information is not retained after type-instantiation. As a result, Java generics cannot support important generic code patterns and declarations such as `T element = new T();` or `class Serial<T> extends T` or `T.class`, where `T` is a type parameter. To address the limitations of erasure, some solutions use clever tricks based on reflection following a library approach [5] but with cumbersome syntax or others extend directly the `javac` compiler [11].

The alternative to erasure is an *expansion*-based translation, where the generic code is replicated at every type-instantiation site. This addresses the expressiveness shortcomings of erasure, but at the cost of replicating generic code. `C++` templates are translated by expansion and they also suffer from another significant drawback: type-checking and code generation is not performed on the generic code but only after expansion, separately for each type instantiation. The Pizza language [8], which is often considered the predecessor and inspiration of the Java generics mechanism, defined both erasure and expansion as possible translation strategies for generics.

In this paper we propose a preliminary language design that combines erasure and expansion of generics at a fine granularity. Each type parameter can specify whether it is to be erased or not. For instance, a generic class `C` can be defined as:

```
class C<@reify X,Y> { ... }
```

In this case, type parameter `Y` is to be erased, while type parameter `X` will not be. As can be seen, type parameters are selected for expansion by use of a Java type annotation. In total, the new elements of our approach are as follows:

- A fine grained language design for controlling expansion or erasure of generic type parameters.
- An extension of the Java language entirely using advanced (upcoming, in Java 8–JSR 308) annotations and the Checker Framework. This is a case study of the expressiveness of these facilities for language extensions that are both fundamental and obtrusive (i.e., the presence of an annotation changes the semantics of code).
- A translation technique that performs expansion, yet strives to minimize code replication by also leveraging delegation.

Our work is currently in the design stage, with full implementation to follow. However, we have conducted preliminary feasibility studies and have high confidence that the mechanics of the extension to the language is realizable as described, using the advanced annotations of JSR 308 and the Checker Framework.

2. Background

We next discuss some necessary background for our work. First, we introduce the concept of mixins, which we will use as a motivating example and demonstration of non-erased type parameters throughout the paper. Next, we present the JSR 308 Java Type Annotations and the Checker Framework—both essential parts of our proposal.

Mixin-Based Programming. (Template-based) mixins are an important, well-known pattern in generic programming. Mixins allow programmers to express component-based designs with clean

GPCE '13, October 27–28, 2013, Indianapolis, Indiana, USA.

ACM 978-1-4503-2373-4/13/10. . . $15.00.
http://dx.doi.org/10.1145/2517208.2517223

and concise class-based modules. A mixin is a standalone entity that can be composed with other entities or mixins, thereby enabling modular behavior sharing. A typical declaration of a mixin class in most OO languages is: `class M<T> extends T`. That is, the mixin generic class, `M`, inherits from its unknown type parameter, `T`. Such a mixin, `M`, is often called an *abstract subclass*. Mixins can be combined to form other mixins and can be composed with other classes. The mixin pattern is one that has appeared many times in the literature—e.g., [2, 12–15]. Mixins cannot be supported when generics are subject to type erasure such as in Java programs and adding mixin support to Java is a non-trivial task [4]. Our proposal is not the first to enable mixin support in Java. However, mixins are simply a use case for demonstrating our more general selective reification approach that permits fine-grained control over expansion and erasure of generics.

Use of Type Annotations and the Checker Framework. The recent addition of JSR 308 permits annotating any use of a type, such as generic type arguments, casts and type declarations. The Checker Framework [3, 9] is an abstraction layer built on top of JSR 308. This framework allows the implementation of our proposal employing Java annotations and integrating a pluggable type system and code generator/transformer. Our language extensions are intrusive to standard Java: the program does not have the same meaning (and in fact may not even compile) when our annotations are erased from the source code. The main advantage of the Checker Framework is that it automatically propagates annotations to compilation units where no annotation appears, thereby enabling type checking and code generation for these units. Additionally, it allows us to use a uniform programming model for AST transformations. A similar technique has also been employed in EnerJ by Sampson et al. [10].

3. Safe Reification for Java Generics

The standard type-erasure semantics of Java generics ensures that the run-time behavior of programs does not depend on type information. This abstraction principle allows generic classes to be compiled exactly *once*, allowing client classes to reuse the *same* bytecode with distinct compile-time type instantiations. However, the type erasure property prohibits the use of mixins. On the contrary, the lack of type erasure in `C++` templates enables the use of mixins, but sacrifices separate compilation and implementation abstraction, as well as leads to code duplication. Our proposal reconciles these two approaches, by enabling selective reification and mixins in a modular manner.

Selective reification. Our proposal preserves the default behavior and properties of generic types: the programmer must explicitly state that a type variable must persist, otherwise the type variable is subject to type erasure. Therefore, we maintain backwards compatibility with standard Java code and permit selective code generation or transformation for types instantiating reified type variables.

The following example illustrates a class definition having two generic variables `X` and `Y` respectively. However, only the latter type variable, `Y`, is subject to type erasure. `X` is not erased, thus, for example, it can be passed to a `new` operator to create a new instance of `X`. Notice that `classOfX` will be instantiated to the object representing the class type instantiating type variable `X`. In the case of `ReifiedGeneric<String,Integer>`, `classOfX` is equal to `String.class`.

```
1  class ReifiedGeneric<@reify X,Y> {
2    Class classOfX = X.class;
3    Y id(Y y) { return y; }
4    X newInstance() { return new X(); }
5  }
```

Mixin support. Type variables annotated with `@reify` can be used within an extends clause, thereby enabling the formulation of mixin classes. The following example illustrates the mixin classes `Serial` and `TimeStamped`. The `Serial` mixin class declaration is equivalent to what we can already express in `C++` with `template<class T> class Serial : public T`. In method `placeOrder`, the `customer` object is an instance of the bottom class in the following hierarchy: `TimeStamped`◁`Serial`◁`Customer`, where ◁ is a shorthand for "`extends`". Therefore, `customer` has the functionality of `Customer` along with a unique serial number and a timestamp indicating its creation time.

```
1  class Serial<@reify T> extends T {
2    static long counter = 1;
3    long serialN = counter++;
4    public long getSerialNumber() {
5      return serialN;
6    }
7  }
8
9  class TimeStamped<@reify T> extends T {
10   long timestamp = new Date().getTime();
11   public long getTimestamp() {
12     return timestamp;
13   }
14 }
15 ...
16   void placeOrder(){
17     TimeStamped<Serial<Customer>> customer =
18       new TimeStamped<Serial<Customer>>();
19     long x = customer.getSerialNumber();
20     long y = customer.getTimestamp();
21     ...
22     customer.order();
23   }
```

4. Safe and modular type system

Our main goal is to enable *modularly safe* type-checking for our language extension: *code generation of a valid program with reified types can never fail*. The inclusion of reification as a language feature requires careful treatment of type variables and type instantiation so as to preserve modular type checking. We informally describe the constraints enforced by our proposed type checker in order to guarantee validity. Let us define class `GenericFactory` as follows:

```
1  class GenericFactory<@reify X> {
2    X newInstance() { return new X(); }
3  }
```

Reified type invariants. Substituting a standard type variable for a reifiable type variable is an invalid operation as there is insufficient type information for performing code generation. For instance allowing an ordinary type variable `Y` to indirectly flow to the instantiation point of `GenericFactory` will inevitably lead to code generation failure and thus violate our safety invariant. The same argument applies when substituting *interfaces* or *abstract* classes where reified types are expected. In order to guarantee modular type checking, we place the restriction that type variables annotated as `@reify` can only be instantiated with another `@reify` type variable or a concrete type. In the later case, the type must provide a public constructor accepting no arguments.

Mixin invariants. The above constraints are sufficient for guaranteeing modular reified type validity. However, they are insufficient when inheritance polymorphism (i.e., mixins) is introduced to the language. Let us assume that `GenericFactory<String>` indirectly flows to `ObjectFactory` through type instantiations.

```
class ObjectFactory<@reify X> extends X {
  Object newInstance() {
    return new Object();
  }
}
```

`ObjectFactory<GenericFactory<String>>` is not a well-formed Java type as there exist two *overloaded* methods `newInstance` with distinct return types `Object` and `String` respectively.

To address non-modular type errors emerging from the propagation of invalid mixin instantiations, we employ *structural* type constraints over reified type variables appearing in `extend` clauses. Structural constraints are expressed in terms of ordinary nominal types `T` at the definition of reified type variables. More specifically, the reified type variable declaration `@reify(T.class) X` denotes that `T<X> extends X` is a well-formed type for any `X`. Any class having a subset of the methods implied by `T<X>` is a structural supertype of `T<X>`. Thus, a structural type `T<X>` acts merely as a macro definition for describing a set of methods rather than as a nominal type. Consequently, `@reify(T.class) X` is implied by `@reify(T′.class) X`, when `T<X>` is a structural supertype of `T′<X>`. Notice that only a constrained reified type can instantiate another constrained type variable. We restrict types representing structural constraints to standard Java types without reification.

```
interface Constraint<X> {
  Object newInstance();
  Long getTimestamp(String);
}
class ObjectFactory<@reify(Constraint.class) X>
extends X {
  Object newInstance() {
    return new Object();
  }
}
```

In the example above, `X` is constrained by `Constraint<X>`, which is a structural supertype of `ObjectFactory<X>`: if any `X` implementing `Constraint<X>` is a well-formed type, then `X` extended with `ObjectFactory<X>` is also well-formed. Using nominal types as macros for structural constraints allow us to minimize the amount of annotations required and makes our constraint specifications modular: it suffices to alter the definition of `Constraint<X>` in order to add constraints as opposed to having to perform explicit inter-module modifications of `@reify` constraints. Finally, when a type parameter `X` is not constrained, we issue a compiler warning (which the user can ignore at her own risk): type safety is not guaranteed modularly.

5. Code Generation

Once the type checking stage is complete, AST transformations are performed in order to lower generic object allocation, type variable type information and mixins to standard Java. The non-trivial part of our translation is that it tries to combine traditional expansion with reuse of the generic's code: most functionality is translated into indirect calls, delegating to a common implementation, rather than always expanding the code in-place. The feasibility of this approach in a full language setting (which, notably, includes overloading resolution) will be a challenge in our future work.

5.1 Transforming classes with reified generics

The transformation process of reified classes entails the erasure of reification annotations from the original class, substitution of type variable allocation expressions with method invocations and declarations and finally a unique interface that contains all methods of the translated class. This interface must be implemented by the translated class. The following example shows the code emitted by the transformation process of our earlier `GenericFactory` class:

```
interface iface$GenericFactory<X> {
  X new$X();
  X newInstance();
}
class GenericFactory<X>
implements iface$GenericFactory<X> {
  X new$X() { return null; }
  X newInstance(){ return new$X(); }
}
```

A new method `new$X()` with no implementation and a subsequent invocation replace the original allocation expression `new$X()`. A new interface `iface$GenericFactory` is generated containing all methods implied by the original class in addition to the generated methods. Class `GenericFactory` implements `iface$GenericFactory` and has no reification annotations.

There are two alternatives for instantiating reified type variables:

Instantiation with other reified variables. In this case, types containing reified type variables are substituted for the generated interfaces described above. A class instantiating a reified type with a type variable `X` has no information regarding `X` except its upper bound, which may be included in the generated interface signature (i.e. in the case of mixins), thus it is safe to substitute the instantiated reified type with its instantiated interface. Similar arguments apply to the environment of the class containing the reified type, thus the generated interface is also employed when accessing class members (i.e. no downcasts are required).

Instantiation with nominal types. Reified variable instantiations with concrete types are transformed to a new class, where reified variables have been substituted for the concrete type. The following example illustrates the generated class code corresponding to type `GenericFactory<Integer>` in the original source program:

```
class GenericFactory$Integer
extends GenericFactory<Integer> {
  Integer new$X() {
    return new Integer();
  }
}
```

The generated class `GenericFactory$Integer` extends `GenericFactory<Integer>` and overrides method `new$X`, which is unimplemented, by returning an allocation expression on the concrete type instantiating `X`.

Translation of fields. Field member declarations are translated by adding getter and setter methods to the generated interface. Field accesses are substituted for invocations to appropriate interface methods.

5.2 Mixins and constrained reification.

Let us modify the `GenericFactory` of the previous example to be a mixin and declare an interface `Constraint` as a structural supertype of `GenericFactory`:

```
interface Constraint<X> {
  X newInstance();
}
class GenericFactory<@reify(Constraint.class) X>
extends X {
  X newInstance(){ return new X(); }
}
```

The transformation process is then modified accordingly.

Instantiation with other reified variables. As in the case of simple reified types, constrained reified types of the form `GenericFactory<Y>` are replaced by the generated interfaces instantiated with

the same type variable `Y`. However, the environment of the class containing a mixin instance may be aware of its concrete parent, therefore downcasts from the generated interfaces to the expected types are performed.

Instantiation with nominal types. Assume we generate code for `GenericFactory<Integer>` as in the previous example. The code generation for `GenericFactory` and `iface$GenericFactory` are identical to the previous example. Let us rename the class of `GenericFactory$Integer` of the previous example to `GenericFactory$$Integer` in order to use it here. The code generation for `GenericFactory<Integer>` differs:

```
class GenericFactory$Integer extends Integer
implements iface$GenericFactory<Integer> {
  iface$GenericFactory<Integer> mixin  =
          new GenericFactory$$Integer();
  Integer new$X() {
    return mixin.new$X();
  }
  Integer newInstance() {
    return mixin.newInstance();
  }
}
```

The key difference is that the generated class is a subtype of both `Integer` and `iface$GenericFactory` and employs delegation on the *mixin* instance in order to avoid code duplication (i.e., in-place substitution of the mixin). Thus, although every mixin is expanded once per instantiation, the actual code content of a mixin (i.e., the bodies of methods) is only generated once.

6. Related work

NextGen [11] retains parametric type information of generics at runtime to support type dependent operations supported by a custom class loader that generates template instantiations at runtime. Our language extension is merely a pluggable module to the `javac` compiler. Additionally in our work we avoid replication by a compile-time generational step to wire class instantiations with mixin code implementations.

Scala [7] supports limited selective reification (but no "abstract-subclass" style mixins) with runtime reflection, a mechanism based on a combination of implicit parameters and `TypeTag` objects carrying implicit type information.

There are many independent projects that overcome the limitations of type erasure. Similarly to our approach, Gafter [5] preserves generic types by declaring anonymous classes at instantiation points. Other works aim to support mixins directly as a language feature by extending Java. Java Layers [2] supports mixins by adopting a source-to-source approach from a custom compiler to valid Java. Jam [1] adopts the same implementation strategy as Java Layers, but suffers from code duplication. Both Java Layers and Jam suffer from non-modular compilation. McJava [6] is an extension of Java allowing for mixin composition with classes and other mixins. McJava does not support modular type checking as mixin well-formedness is validated at type instantiation without imposing intra-procedural constraints. McJava code generation performs copying of mixin declarations to instantiation points. Our approach only generates code for each mixin *once* and performs wiring with specific instantiations, thereby avoiding code duplication.

7. Summary

In this work, we presented the `@reify` annotation that extends Java generics with by-expansion translation relative to selected type parameters only. We demonstrate the translation schemes of classes having both reified and standard generic parameters. We support mixins and allocation expressions using generic types in a modular manner. The logic behind type checking and generation is based on a pluggable type system, without modifying the Java compiler.

Acknowledgments

We gratefully acknowledge funding by the Greek Secretariat for Research and Technology under the "MorphPL" Excellence (Aristeia) award; and by the European Union under a Marie Curie International Reintegration Grant and the "SPADE" European Research Council Starting/Consolidator grant.

References

[1] D. Ancona, G. Lagorio, and E. Zucca. Jam-Designing a Java Extension with Mixins. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 25(5):641–712, 2003.

[2] R. Cardone, A. Brown, S. McDirmid, and C. Lin. Using Mixins to Build Flexible Widgets. In *Proceedings of the 1st International Conference on Aspect-Oriented Software Development (AOSD)*, pages 76–85, New York, NY, USA, 2002.

[3] W. Dietl, S. Dietzel, M. D. Ernst, K. Muşlu, and T. W. Schiller. Building and using Pluggable Type-Checkers. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 681–690, 2011.

[4] B. Eckel. Mixins: Something Else You Can't Do With Java Generics? `http://www.artima.com/weblogs/viewpost.jsp?thread=132988`, Oct. 2005.

[5] N. Gafter. Super Type Tokens. `http://gafter.blogspot.gr/2006/12/super-type-tokens.html`, Dec. 2006.

[6] T. Kamina and T. Tamai. McJava - A Design and Implementation of Java with Mixin-Types. In *Proceedings of the Programming Languages and Systems: Second Asian Symposium, (APLAS)*, pages 4–6, Taipei, Taiwan, 2004.

[7] M. Odersky. The Scala Language Specification, 2013.

[8] M. Odersky and P. Wadler. Pizza into Java: translating theory into practice. In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of Programming languages (POPL)*, POPL '97, pages 146–159, New York, NY, USA, 1997. ACM.

[9] M. M. Papi, M. Ali, T. L. Correa,Jr., J. H. Perkins, and M. D. Ernst. Practical Pluggable Types for Java. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis*, ISSTA '08, pages 201–212, New York, NY, USA, 2008. ACM.

[10] A. Sampson, W. Dietl, E. Fortuna, D. Gnanapragasam, L. Ceze, and D. Grossman. EnerJ: approximate data types for safe and general low-power computation. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI)*, pages 164–174, San Jose, California, USA, 2011.

[11] J. Sasitorn and R. Cartwright. Efficient First-Class Generics on Stock Java Virtual Machines. In *Proceedings of the 2006 ACM symposium on Applied computing*, SAC '06, pages 1621–1628, New York, NY, USA, 2006. ACM.

[12] Y. Smaragdakis and D. Batory. Implementing Layered Designs with Mixin Layers. In *Proceedings of the 12th European Conference on Object-Oriented Programming (ECOOP)*, pages 550–570. Springer-Verlag LNCS 1445, 1998.

[13] Y. Smaragdakis and D. Batory. Mixin-Based Programming in C++. In *Generative and Component-Based Software Engineering Symposium (GCSE)*, pages 163–177. Springer-Verlag, LNCS 2177, 2000.

[14] M. VanHilst and D. Notkin. Using C++ Templates to Implement Role-Based Designs. In *JSSST International Symposium on Object Technologies for Advanced Software*, pages 22–37. Springer Verlag, 1996.

[15] M. VanHilst and D. Notkin. Using Role Components to Implement Collaboration-Based Designs. In *Proceedings of the 1996 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pages 359–369, San Jose, California, USA, 1996.